described in [0037] to [0039] is the location in the task class mirror table of the initialized and resolved entries for a given task.

Initialization

[0042] FIG. 4 is a flowchart illustrating how a multitasking virtual machine implements access to a global variable of a class by one task in accordance with one embodiment of the present invention. FIG. 4 describes two types of access to a global variable of a class: accesses guarded by a class initialization barrier, and unguarded accesses. Guarded accesses are used in the implementation of the interpreter of the multitasking virtual machine, and are generated by its dynamic compiler. Unguarded accesses are typically generated by the dynamic compiler of a multitasking virtual machine, when the dynamic compiler determines through some analysis that a class initialization barrier is unnecessary. In both cases of access, the flowchart assumes that the offset to the static variable being accessed is already set in a register Roffset.

[0043] A guarded class variable access starts with setting a register to the address of the task class mirror table associated with the class whose variable is being accessed (step 400). It then fetches the encoded identifier of the current task from the current thread's descriptor, which is typically bound to a specific register (step 402). The value of the encoded identifier is an offset from the beginning of the task class mirror table to the initialized entry associated with the current task. The registers set by the two previous steps are then used to load a pointer to a task class mirror object into a register R3 (step 404), the value of which is tested (step 406). A non-null pointer value indicates that the class has been initialized, and control is transferred to code that load the class variable (step 410).

[0044] A null pointer value in R3 indicates that the class whose variable is being accessed has not been initialized yet, and control is transferred (step 406) to code that calls a runtime function (step 408) that performs the initialization of the class, if needed. The runtime function synchronizes all the threads of the current task so that only one of them, called the initializer thread, executes the class initialization code. While executing the class initialization code, the initializer thread may execute further guarded accesses to the variables of the class being initialized. These guarded accesses too result in a call to the runtime function, since the initialized entry in the task class mirror table is still set to null. Any call to the runtime from a guarded access to a variable of a class C that is already being initialized, and issued by the initializer thread of C, will return immediately without further action. However, because the initialized entry is still set to null, the initializer thread cannot use it to access the class's variable. Instead, upon return from the call to the runtime at step 408, execution continues as for an unguarded class variable access (steps 420, 422, 424, 410). An equivalent alternative to this solution is to have the runtime function at step 408 returning the pointer to the task class mirror object and to proceed directly to step 410.

[0045] An unguarded class variable access starts with setting a register to the address of the task class mirror table associated with the class whose variable is being accessed (step 420). It then fetches the encoded identifier of the current task from the current thread's descriptor, which is typically bound to a specific register (step 422). The value of the encoded identifier is an offset from the beginning of the task class mirror table to the resolved entry associated with the current task. The registers set by the two previous steps are then used to load a pointer to a task class mirror object into a register R3 (step 424), which is then used to load the desired class variable (step 410). Multiple variants of this mechanism are possible. For instance, it is possible to compute the offset to the resolved entry from the offset to the initialized entry (as described in [0054]) instead of storing the former in thread descriptors.

[0046] If the implementation of the multitasking virtual machine treats initialization-less classes specially in order to optimize space consumed by task class mirror table, all accesses to the variables of an initialization-less class can be unguarded. However, unguarded accesses in this case differ from the description given FIG. 4 as follows: step 422 should load the initialized encoded task identifier (as in 402) instead of the resolved one, since task class mirror tables associated with initialization-less classes does not have any resolved entries. Note also that guarded accesses, as described on FIG. 4, to the static variables of an initialization-less class always works, since in this case the register R3 will never be null. Guarded access to initialization-less classes is however sub-optimal and shouldn't be used if the multitasking virtual machine implementation adopt the special task class mirror table arrangement for initialization-less classes.

[0047] The following illustrates in detail how the class initialization barrier and static variable access mechanisms described above can be implemented on a particular processor, namely, SPARC v9. SPARC is a trademark or registered trademark of SPARC International, Inc. A guarded access to a static variable is typically implemented with the following sequence of instructions:

```
1.      ld [gthread + encoded_task_id], initialized_entry_offset
2.      ld [tcm_table + initialized_entry_offset], tcm
3.      brnz, pt, a, tcm end_barrier
4.      ld [tcm + static_var_offset], tmp
5.      call task_class_initialization_stub
6.      nop
7.      ld [tcm + static_var_offset], tmp
8.      end_barrier:
```